Lesson 2

Christian Schwarz, Jakob Krebs

04.11.2019

Contents

Source Code and Solutions

Sources and Solutions

- we publish all code written in this course at `https://github.com/jkrbs/c_lessons`
- we will publish example solutions of the tasks on same site
- send us questions or your solutions to c-lessons@deutschland.gmbh

Variables and Types

Integers

- Keywords: `int`, `short`, `long`, `long long`
- Stored as a binary number with fixed length
- Can be `signed` or `unsigned` (undefined, but can be overridden using signed char and unsigned char)
- Actual size of `int`, `short`, `long` depends on architecture
- For definite sizes: include stddef.h which adds types like `size_t`, `int32_t`, `uint64_t`

Floating Point Numbers

- Keywords: `float`, `double`, `long double`
- Stored as specified in *IEEE 754 Standard* TL;DR
- Special values for ∞, $-\infty$, NaN
- Useful for fractions and very large numbers
- Type a decimal point instead of a comma!

Example:

```c
float x = 0.125;             /* Precision: 7 to 8 digits */
double y = 111111.111111;    /* Precision: 15 to 16 digits */
```

Characters

- Keyword: `char`
- Can be `signed` (default) or `unsigned`
- Size: 1 Byte (8 Bit) on almost every architecture
- Intended to represent a single character
- Stores its *ASCII* number (e.g. 'A' ⇒ 65)

You can define a `char` either by its ASCII number or by its symbol:

```c
char a = 65;
char b = 'A';    /* use single quotation marks */
```

format strings

format strings

The format string determines how a value is interpreted in the `printf` function family. Here are some of the available options:

type	**description**	**type of argument**
%c	single character	char, int (if $<=$ 255)
%d or %i	decimal number	char, int
%u	unsigned decimal number	unsigned char, unsigned int
%X	hexadecimal number	char, int
%ld	long decimal number	long
%f	floating point number	float, double
%s	string	const char* [more on this later]

printf and scanf

printf and scanf

We already know `printf` allows us to write out data to the console.

`scanf` does the opposite, and reads in user input from the console:

```c
puts("Please insert a number:");
int number;
scanf("%d", &number); //reads in a single number

char c;
//reads in a number and a char separated by whitespace
scanf("%d %c", &number, &c);
```

`scanf` actually returns an int. That is the number of successfully read arguments.

The `&number` means ”place the read result into the number variable”. Treat it as magic for now, we will explain it properly later.

Operators

Basic Binary Operators

- `+`, `-` just behave as expected
- `*` means multiply, `/` means divide
- Operator precedence works mostly as expected.
- You can use parenthesess around expressions: `(3 + 4) * 7`
- `=` is the assignment operator.
  - `x = 4;` means that future references to `x` will evaluate to `4`
  - you cannot assign to arbitrary expressions: `(x + 1) = 17` is not legal, since `(x + 1)` is not assignable. A compile time error occurs.
- `==` is the comparison operator. `4 == 4` evaluates to `true`, `x * 0 == 1` evaluates to `false`
- `%` is the modulus operator. Examples: `7 % 3 == 1`, `2 % 2 == 0`

logical operators and comparisons

- `<` less than
- `<=` less or equal than
- `>` greater than
- `>=` greater or equal than
- `&&` and
- `||` or
- `!` negation

bitwise operations

a	b	$a \mid b$	$a\&b$	$a \wedge b$
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

`5 ^ 3 == 6`

$$0101 \oplus 0011 = 0110 \equiv 6$$

Type Conversions

- Explicit type conversion can be performed using the casting syntax:

```c
int i = 5;
float fi = (float)i;
```

- When mixing different types in an expression, C will convert the types to match. The rules applying here are rather complicated, please use explicit casts instead like this:

```c
int i = 5;
float res = (float)i * 3;
```

- Be especially wary of mixing `signed` and `unsigned` integers!

Control Structures

if statements

- basic usage:

```
if(3 > 2){  // arbitrary condition
    //this gets executed IF the condition evaluates to true
}
```

- short form (use it only for short and simple things):

```
if(3 > 2) bar();
```

- else blocks:

```
if(foo()){
}
else if(bar()){
}
else{
}
```

what we really use here is the shorthand notation on the else block

basic while statements

```c
int i = 0;
while(i < 20){
    printf("%i\n", i);
    i++;
}
```

This loop prints the numbers from 0 to 19(inclusive). Before each iteration (even before the first) the condition is checked. Once the condition is no longer satisfied, it jumps after the loop block.

break and continue in while statements

```
int i = 0;
while(true){
    i++;
    if(i % 7 == 0) continue; //skip all numbers divisible by 5
    printf("%i\n", i);
    if(i == 20)break; //exit the loop once i is 20
}
```

- `continue` skips the rest of the loop block and begins the next iteration
- `break` just jumps after the end of the loop block
- Beware: if you have a `switch` inside a `while`, `break` will just exit the `switch`!
- In fact, `break` and `continue` will always be applied to the innermost control structure that defines them.

do...while

The difference between `do...while` and `while` is the order of executing the statement(s) and checking the condition.

The `while` loop begins with checking, while the `do...while` loop begins with executing the statement(s).

```
int i = 3;
do {
    - -i;
} while (i < 1);
```

The

Statement(s) in a `do...while` loop are executed at least once.

for

The For-Loop is comfortable for iterating. It takes three arguments.

- Initialization
- Condition
- Iteration statement

For illustration, consider a program printing the numbers 1 to 10:

```c
for (int i = 1; i <= 10; ++i){
    printf("%d\n", i);
}
```

- *i* is called an *index* iterating from the given start to a given end value
- *i*, *j*, *k* are commonly used identifiers for the index

switch statements

Switch statements are useful when you have lots of different `if` cases and know all possible cases at compile time.

```c
switch(command_that_returns_a_status_code()){
    case 0: break; //everything is ok
    // missing break! fallthrough! (or intended??)
    case 1: puts("we ran out of disk space");
    case 17: puts("foo"); break;
}
```

Depending on the result of the function, the switch jumps to the respective `case`. Every `case` must be terminated by a `break;` statement, otherwise the following `case`(s) also get executed. If this is really your intention, which happens very rarely, put a comment like `//fallthrough`, since this is a common bug.

switch statements 2

```
    switch(foo()){
        case 0: puts(" :)"); break;
        case 2: {
            puts("some logging output");
            puts("more logging output");
        }break;
        default: puts("this should never happen(TM)");
    }
```

`case` bodys can be blocks. Remember that you still need a break after the block though! The `default` case gets used if no other once matches. If it is the last case, you may leave out the `break`.

Functions

Functions

A regular function has a return type, a name, parameters and a body

```c
int add(int a, int b){
    printf("%i + %i = %i\n", a, b, a + b);
    return a + b;
}
```

`printf` is also a function but the number of its arguments can vary (`varargs`). we will talk about this later.

Void Functions

- A function can also return nothing, the type of ”*nothing*” is `void`.
- `void` returning functions should not contain `return` statements
- Functions can call other functions (including themselves, which is called recursion)
- A function with no parameters should have `(void)` instead of `()` as it’s parameter specification, as C will otherwise treat the numer of parameters as undefined

```c
void foo(void){
    puts("I’m a very boring function :(");
}
```